JAVA

- 컬렉션 프레임워크

제네릭 클래스의 이해와 설계

AppleBox와 OrangeBox 클래스의 설계

```
class AppleBox{
      Apple item;
      public void store(Apple item) { this.item=item; }
      public Apple pullOut( ) { return item; }
}

class OrangeBox{
      Orange item;
      public void store(Orange item) { this.item=item; }
      public Orange pullOut( ) { return item; }
}
```

```
class FruitBox{
      object item;
      public void store(Object item) { this.item=item; }
      public Object pullOut( ) { return item; }

}
```

AppleBox와 OrangeBox 클래스의 설계

구현의 편의만 놓고 보면, FruitBox 클래스가 더 좋아 보인다.
하지만 FruitBox 클래스는 자료형에 안전 하지 못하다는 단점이 있다.

물론 AppleBox와 OrangeBox는 구현의 불편함이 따르는 단점이 있다.
그러나 자료형에 안전하다는 장점이 있다.

AppleBox, OrangeBox의 장점인 **자료형의 안전성과** FruitBox의 장점인
구현의 편의성을 한데 모은 것이 **제네릭이다!**

AppleBox와 OrangeBox 클래스의 설계

```
class Orange
{
	int sugarContent;		// 당분 함량
	public Orange(int sugar) {
		sugarContent=sugar;
	}
	public void showSugarContent()
	{
		System.out.println("당도 "+sugarContent);
	}
}

class FruitBox
{
	Object item;
	public void store(Object item) { this.item=item; }
	public Object pullOut() { return item; }
}
```

AppleBox와 OrangeBox 클래스의 설계

```
class ObjectBaseFruitBox
{
        public static void main(String[] args)
        {
                FruitBox fBox1=new FruitBox();
                fBox1.store(new Orange(10));
                Orange org1=(Orange)fBox1.pullOut();
                org1.showSugarContent();

                FruitBox fBox2=new FruitBox();
                fBox2.store("오렌지");
                Orange org2=(Orange)fBox2.pullOut();
                org2.showSugarContent();
        }
}
```

자료형의 안전성에 대한 논의

실행 중간에 Class Casting Exception이 발생한다!

그런데 실행 중간에 발생하는 예외는 컴파일 과정에서 발견되는 오류 상황보다 발견 및 정정이 어렵다!

즉, 이는 자료형에 안전한 형태가 아니다.

자료형의 안전성에 대한 논의 예제

```
class Orange
{
        int sugarContent;          // 당분 함량
        public Orange(int sugar) { sugarContent=sugar; }
        public void showSugarContent() {
                System.out.println("당도 "+sugarContent);
        }
}

class OrangeBox
{
    Orange item;
    public void store(Orange item) { this.item=item; }
    public Orange pullOut() { return item; }
}
```

자료형의 안전성에 대한 논의 예제

```
class OrangeBaseOrangeBox
{
        public static void main(String[] args)
        {
                OrangeBox fBox1=new OrangeBox();
                fBox1.store(new Orange(10));
                Orange org1=fBox1.pullOut();
                org1.showSugarContent();

                OrangeBox fBox2=new OrangeBox();
                fBox2.store("오렌지");
                Orange org2=fBox2.pullOut();
                org2.showSugarContent();
        }
}
```

자료형의 안전성에 대한 논의

컴파일 과정에서, 메소드 store에 문자열 인스턴스가 전달될 수 없음이 발견된다.

이렇듯 컴파일 과정에서 발견된 자료형 관련 문제는 발견 및 정정이 간단하다.

즉, 이는 자료형에 안전한 형태 이다.

제네릭(Generics) 클래스 설계

```
class FruitBox
{
      object item;
      public void store(Object item) { this.item=item; }
      public Object pullOut( ) { return item; }

}
```

// <T> → T라는 이름이 매개 변수화된 자료형임을 나타냄

```
class FruitBox <T>
{
      T item;
      public void store(T item) { this.item=item; }
      public T pullOut( ) { return item; }

}
```

T에 해당하는 자료형의 이름은 인스턴스를 생성하는 순간에 결정이 된다!

제네릭 클래스 기반 인스턴스 생성

class FruitBox<Orange> orBox=new FruitBox <Orange>();

T를 Orange로 결정해서 FruitBox의 인스턴스를 생성하고 이를 참조할 수 있는 참조 변수를 선언해서 참조 값을 저장한다!

class FruitBox<Apple> apBox=new FruitBox <Apple>();

T를 Apple로 결정해서 FruitBox의 인스턴스를 생성하고 이를 참조할 수 있는 참조 변수를 선언해서 참조 값을 저장한다!

제네릭 클래스 기반 인스턴스 생성

```
public static void main(String[] args)
{
        FruitBox<Orange> orBox=new FruitBox<Orange>();
        orBox.store(new Orange(10));
        Orange org=orBox.pullOut();
        org.showSugarContent();

        FruitBox<Apple> apBox=new FruitBox<Apple>();
        apBox.store(new Apple(20));
        Apple app=apBox.pullOut();
        app.showAppleWeight();
}
```

인스턴스 생성시 결정된 T의 자료형에 일치하지 않으면 컴파일 에러가 발생하므로 자료형에 안전한 구조임!

제네릭(Generics) 클래스 설계 예제

```
class Orange
{
	int sugarContent;		// 당분 함량
	public Orange(int sugar) { sugarContent=sugar; }
	public void showSugarContent()
	{
		System.out.println("당도 "+sugarContent);
	}
}

class Apple
{
	int weight;		// 사과의 무게
	public Apple(int weight) { this.weight=weight; }
	public void showAppleWeight()
	{
		System.out.println("무게 "+weight);
	}
}
```

제네릭(Generics) 클래스 설계 예제

```
class FruitBox<T>
{
	T item;
	public void store(T item) { this.item=item; }
	public T pullOut() { return item; }
}

class GenericBaseFruitBox
{
	public static void main(String[] args)
	{
		FruitBox<Orange> orBox=new FruitBox<Orange>();
		orBox.store(new Orange(10));
		Orange org=orBox.pullOut();
		org.showSugarContent();

		FruitBox<Apple> apBox=new FruitBox<Apple>();
		apBox.store(new Apple(20));
		Apple app=apBox.pullOut();
		app.showAppleWeight();
	}
}
```

제네릭(Generics) 클래스 설계 예제

문제 1.

예제 GenericBaseBox.java의 FruitBox<T> 클래스에 생성자를 추가하여, 다음의 main 메서드가 컴파일 및 실행됨을 확인해보자.

```
class GenericConstructor  {
        pulic static void main(String[] args) {
        FruitBox<Orange> orBox = new FruitBox<Orange>(new Orange(10));
        Orange org = orBox.pullOut();
        org.showSugarContent();

        FruitBox<Apple> apBox = new FruitBox<Apple>(new Apple (10));
        Apple app = apBox.pullOut();
        app.showAppleWeight();
        }
}
```

참고로 네네릭 클래스의 생성자 정의방법은 일반 클래스 생성자 정의방법과 차이가 없다.

컬렉션 프레임워크

컬렉션 프레임워크의 이해

컬렉션 프레임워크의 기본적인 이해

- 프레임 워크가 의미하는 바는 다음과 같다.

  → 잘 정의된, 약속된 구조와 골격

- 자바의 컬렉션 프레임워크

  → 인스턴스의 저장과 참조를 위해 잘 정의된, 클래스들의 구조

- 컬렉션 프레임워크가 제공하는 기능의 영역

  → 자료구조와 알고리즘

컬렉션 프레임워크의 기본적인 이해

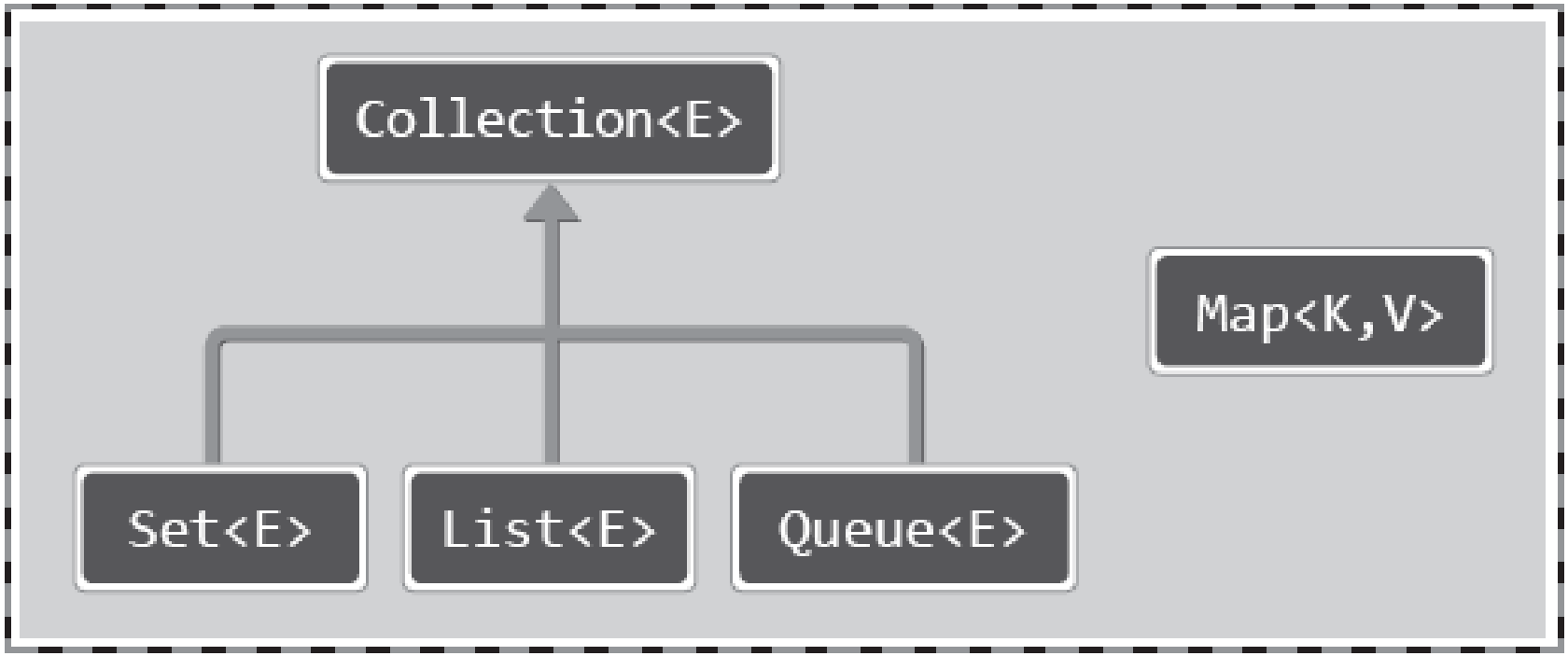
자바의 컬렉션 프레임워크는 별도의 구현과 이해 없이
자료구조와 알고리즘을 적용할 수 있도록 설계된 클래스들의 집합이다.
그러나 자료 구조의 이론적인 특성을 안다면, 보다 적절하고 합리적인 활용이 가능하다.

자료구조 : 배열, 리스트, 스택, 큐, 트리, 해시 등

알고리즘 : 정렬, 탐색, 최대, 최소 등

컬렉션 프레임워크의 기본 골격

* 컬렉션 프레임 워크의 인터페이스 구조



컬렉션 프레임워크의 기본 골격

- **Collection<E> 인터페이스를 구현하는 제네릭 클래스**
  → 인스턴스 단위의 데이터 저장 기능 제공
  (배열과 같이 단순 인스턴스 참조 값 저장)

- **Map<K, V>**
  → key-value 구조의 인스턴스 저장 기능 제공

Collection<E> 인터페이스를 구현하는 제네릭 클래스들

ArrayList<E>, LinkedList<E>

- List<E> 인터페이스를 구현하는 대표적인 제네릭 클래스

  → ArrayList<E>, LinkedList<E>

- List<E> 인터페이스를 구현 클래스의 인스턴스 저장 특징

  → 동일한 인스턴스의 중복 저장을 허용한다.

  → 인스턴스의 저장 순서가 유지된다.

ArrayList<E>, LinkedList<E>

```
public static void main(String[] args)
{
        ArrayList<Integer> list=new ArrayList<Integer>();

        /* 데이터의 저장 */
        list.add(new Integer(11));
        list.add(new Integer(22));
        list.add(new Integer(33));

        /* 데이터의 참조 */
        System.out.println("1차 참조");
        for(int i=0; i<list.size(); i++)      System.out.println(list.get(i));

        /* 데이터의 삭제 */
        list.remove(0);
        System.out.println("2차 참조");
        for(int i=0; i<list.size(); i++)      System.out.println(list.get(i));
}
```

ArrayList<E> 예제

```
import java.util.ArrayList;

class IntroArrayList
{
        public static void main(String[] args)
        {
                ArrayList<Integer> list=new ArrayList<Integer>();
                /* 데이터의 저장 */
                list.add(new Integer(11));
                list.add(new Integer(22));
                list.add(new Integer(33));

                /* 데이터의 참조 */
                System.out.println("1차 참조");
                for(int i=0; i<list.size(); i++) System.out.println(list.get(i));

                /* 데이터의 삭제 */
                list.remove(0);
                System.out.println("2차 참조");
                for(int i=0; i<list.size(); i++)  System.out.println(list.get(i));
        }
}
```

LinkedList<E>

- 데이터의 저장 방식

  → 이름이 의미하듯이 '리스트'라는 자료구조를 기반으로 데이터를 저장한다.

- 사용방법

  → ArrayList<E>의 사용방법과 거의 동일하다!

  다만, 데이터를 저장하는 방식에서 큰 차이가 있을 뿐이다.

- 대부분의 경우 ArrayList<E>를 대체 할 수 있다.

LinkedList<E>

```
public static void main(String[] args)
{
        LinkedList<Integer> list=new LinkedList<Integer>();

        /* 데이터의 저장 */
        list.add(new Integer(11));
        list.add(new Integer(22));
        list.add(new Integer(33));

        /* 데이터의 참조 */
        System.out.println("1차 참조");
        for(int i=0; i<list.size(); i++)
        System.out.println(list.get(i));

        /* 데이터의 삭제 */
        list.remove(0);
        System.out.println("2차 참조");
        for(int i=0; i<list.size(); i++)
        System.out.println(list.get(i));
}
```

LinkedList<E>

```
import java.util.LinkedList;

class IntroLinkedList
{
        public static void main(String[] args)
        {
                LinkedList<Integer> list=new LinkedList<Integer>();
                /* 데이터의 저장 */
                list.add(new Integer(11));
                list.add(new Integer(22));
                list.add(new Integer(33));

                /* 데이터의 참조 */
                System.out.println("1차 참조");
                for(int i=0; i<list.size(); i++) System.out.println(list.get(i));

                /* 데이터의 삭제 */
                list.remove(0);
                System.out.println("2차 참조");
                for(int i=0; i<list.size(); i++)  System.out.println(list.get(i));
        }
}
```

ArrayList<E>와 LinkedList<E>의 차이점

- 저장소의 용량을 늘리는 과정에서 많은 시간이 소요된다.

  ArrayList<E>의 **단점**
- 데이터의 삭제에 필요한 연산과정이 매우 길다.

  ArrayList<E>의 **단점**
- 데이터의 참조가 용이해서 빠른 참조가 가능하다.

  ArrayList<E>의 **장점**

ArrayList<E>와 LinkedList<E>의 차이점

- 저장소의 용량을 늘리는 과정이 간단하다.

  LinkedList<E>의 장점
- 데이터의 삭제가 매우 간단하다.

  LinkedList<E>의 장점
- 데이터의 참조가 다소 불편하다.

  LinkedList<E>의 단점

Iterator를 이용한 인스턴스의 순차적 접근

- Collection<E> 인터페이스에는 iterator라는 이름의 메소드가 다음의 형태로 정의
  → Iterator<E> iterator( ) { . . . . }
- iterator 메소드가 반환하는 참조 값의 인스턴스는 Iterator<E> 인터페이스를 구현하고 있다.
- iterator 메소드가 반환한 참조 값의 인스턴스를 이용하면, 컬렉션 인스턴스에 저장된 인스턴스의 순차적 접근이 가능함.
- iterator 메소드의 반환형이 Iterator<E>이니, 반환된 참조 값을 이용해서Iterator<E>에 선언된 함수들만 호출하면 된다.

Iterator를 이용한 인스턴스의 순차적 접근

- **boolean hasNext( )** 참조할 다음 번 요소(element)가 존재하면 true를 반환
- **E next()** 다음 번 요소를 반환
- **void remove()** 현재 위치의 요소를 삭제

Iterator를 이용한 인스턴스의 순차적 접근

```
public static void main(String[] args)
{
        LinkedList<String> list=new LinkedList<String>();
        list.add("First");
        list.add("Second");
        list.add("Third");
        list.add("Fourth");

        Iterator<String> itr=list.iterator();
        System.out.println("반복자를 이용한 1차 출력과 ₩"Third₩" 삭제");

        while(itr.hasNext()){
                String curStr=itr.next();
                System.out.println(curStr);
                if(curStr.compareTo("Third")==0)          itr.remove();
        }
        System.out.println("₩n₩"Third₩" 삭제 후 반복자를 이용한 2차 출력 ");
        itr=list.iterator();
        while(itr.hasNext())        System.out.println(itr.next());
}
```

반복자를 사용하는 이유

- 반복자를 사용하면, 컬렉션 클래스의 종류에 상관없이 동일한 형태의 데이터 참조방식을 유지할 수 있다.
- 따라서 컬렉션 클래스의 교체에 큰 영향이 없다.
- 컬렉션 클래스별 데이터 참조 방식을 별도로 확인할 필요가 없다.

왼편은 앞서 소개한 예제이다. 그런데, 이 예제는 반복자를 사용했기 때문에, LinkedList<E>가 어울리지 않아서, 컬렉션클래스를 HashSet<E>로 변경해야 할 때, 다음과 같이 변경이 매우 용이하다.

```
LinkedList<String> list = new LinkedList<String>( );
```

```
HashSet<String> set = new HashSet<String>( );
```

컬렉션 클래스를 이용한 정수의 저장

```
ArrayList<int> arr=new ArrayList<int>( );        error

LinkedList<int> link=new LinkedList<int>( );     error
```

기본 자료형 정보를 이용해서 제네릭 인스턴스 생성 불가능!

따라서 Wraper 클래스를 기반으로 컬렉션 인스턴스를 생성한다.

컬렉션 클래스를 이용한 정수의 저장

```
public static void main(String[] args)
{
        LinkedList<Integer> list=new LinkedList<Integer>();

        list.add(10);               // Auto Boxing
        list.add(20);               // Auto Boxing
        list.add(30);               // Auto Boxing

        Iterator<Integer> itr=list.iterator();

        while(itr.hasNext())
        {
                int num=itr.next();                 // Auto Unboxing
                System.out.println(num);
        }
}
```

컬렉션 클래스를 이용한 정수의 저장

```
import java.util.Iterator;
import java.util.LinkedList;

class PrimitiveCollection{
        public static void main(String[] args){
                LinkedList<Integer> list=new LinkedList<Integer>();
                list.add(10);           // Auto Boxing
                list.add(20);           // Auto Boxing
                list.add(30);           // Auto Boxing

                Iterator<Integer> itr=list.iterator();

                while(itr.hasNext())
                {
                        int num=itr.next();     // Auto Unboxing
                        System.out.println(num);
                }
        }
}
```

Set<E> 인터페이스를 구현하는 컬렉션 클래스들

Set<E> 인터페이스의 특성과 HashSet<E> 클래스

- List<E>를 구현하는 클래스들과 달리 Set<E>를 구현하는 클래스들은 데이터의 저장 순서를 유지하지 않는다. → 반복문 x
- List<E>를 구현하는 클래스들과 달리 Set<E>를 구현하는 클래스들은 데이터의 중복저장을 허용하지 않는다. 단, 동일 데이터에 대한 기준은 프로그래머가 정의 (중복의 조건은 프로그래머가 정의할수있다)
- 즉, Set<E>를 구현하는 클래스는 '집합'의 성격을 지닌다.

Set<E> 인터페이스의 특성과 HashSet<E> 클래스

```
public static void main(String[] args)
{
        HashSet<String> hSet=new HashSet<String>();

        hSet.add("First");
        hSet.add("Second");
        hSet.add("Third");
        hSet.add("First");

        System.out.println("저장된 데이터 수: "+hSet.size());

        Iterator<String> itr=hSet.iterator();
        while(itr.hasNext()) System.out.println(itr.next());
}
```

동일 인스턴스의 판단기준 관찰을 위한 예

```
class SimpleNumber
{
        int num;
        public SimpleNumber(int n)
        {
                num=n;
        }
        public String toString()
        {
                return String.valueOf(num);
                //static 메소드인 valueOf는 기본 자료형 데이터를 String 인스턴스로 변환
        }
}
```

HashSet<E> 클래스의 인스턴스 동등 비교방법

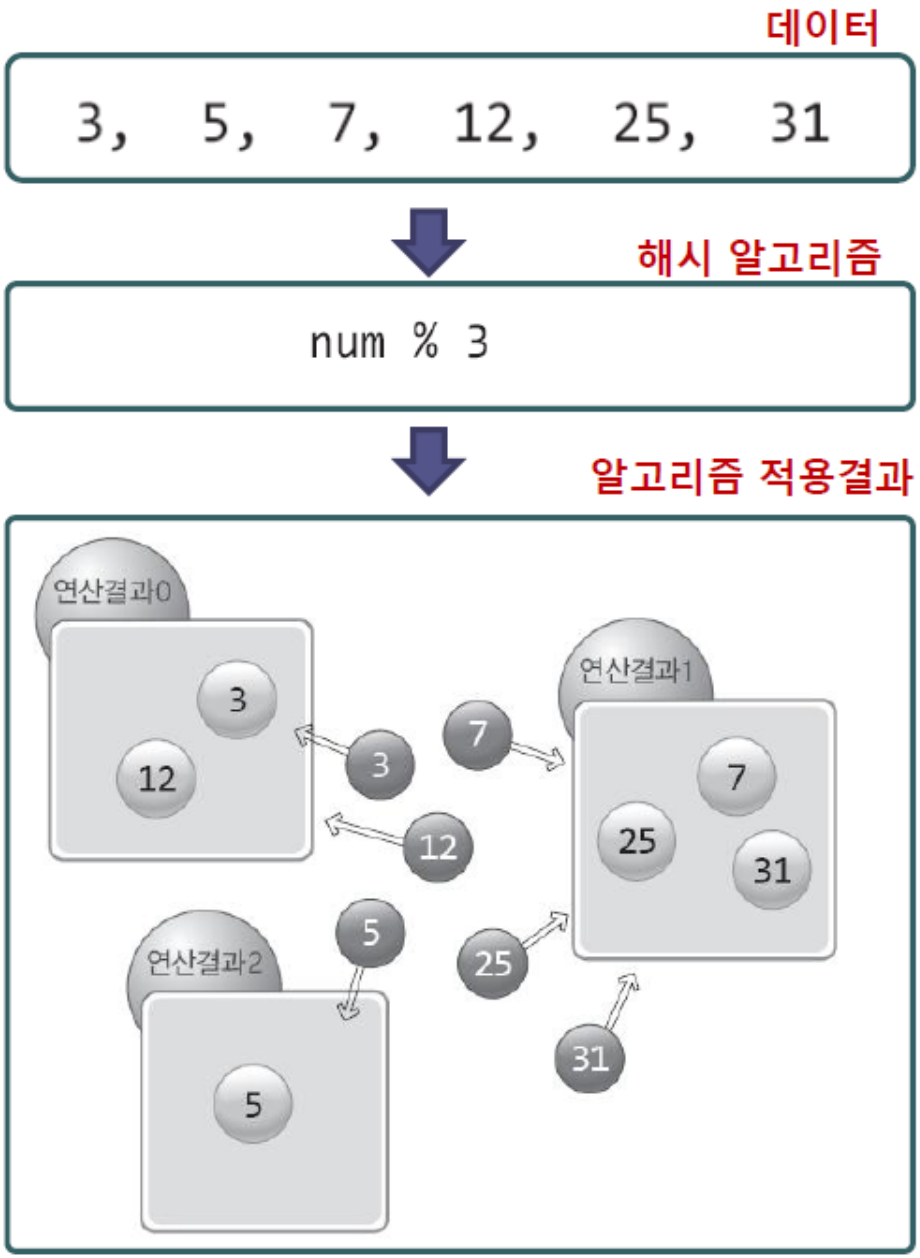
Object 클래스에 정의되어 있는 equals 메소드의 호출 결과와

hashCode 메소드의 호출결과를 참조하여 인스턴스의 동등비교를 진행

동일 인스턴스의 판단기준 관찰을 위한 예

```
public static void main(String[] args)
{
    HashSet<SimpleNumber> hSet=new HashSet<SimpleNumber>();

    hSet.add(new SimpleNumber(10));
    hSet.add(new SimpleNumber(20));
    hSet.add(new SimpleNumber(20));

    System.out.println("저장된 데이터 수: "+hSet.size());

    Iterator<SimpleNumber> itr=hSet.iterator();

    while(itr.hasNext())
        System.out.println(itr.next());
}
```

해시 알고리즘의 이해(데이터의 구분)



해시알고리즘은 이렇듯 간단하게 디자인 될 수도 있다.

해시 알고리즘은 데이터의 분류에 사용이 된다.
데이터를 3으로 나머지 연산하였을 때 얻게 되는 반환 값을 '해시 값'으로 하여 총 세 개의 부류를 구성하였다.

이렇게 분류해 놓으면, 데이터의 검색이 빨라진다.
정수 12가 저장 되어있는지 확인한다고 했을 때 문제 정수 12의 해시 값을 구한다.
그 다음에 해시 값에 해당하는 부류에서만 정수 12의 존재 유무를 확인하면 된다..

HashSet<E> 클래스의 동등 비교

- **검색1단계**
  Object 클래스의 hashCode 메소드의 반환 값을 해시 값으로 활용하여 검색의 그룹을 선택한다.

- **검색2단계**
  그룹내의 인스턴스를 대상으로 Object 클래스의 equals 메소드의 반환 값의 결과로 동등을 판단

HashSet<E>의 인스턴스에 데이터의 저장을 명령하면, 우선 다음의 순서를 거치면서 동일 인스턴스가 저장 되었는지를 확인한다.

따라서 아래의 두 메소드를 적절히 오버라이딩 해야 함.

```
public int hashCode( )
public boolean equals(Object obj)
```

hashCode 메소드의 구현에 따라서 검색의 성능이 달라진다. 그리고 동일 인스턴스를 판단하는 기준이 맞게 equals 메소드를 정의해야 한다.

HashSet<E> 클래스

```
class SimpleNumber{
        int num;
        public SimpleNumber(int n){
                num=n;
        }
        .....

        public int hashCode(){
                return num%3;
        }

        public boolean equals(Object obj){

                SimpleNumber comp=(SimpleNumber)obj;
                if(comp.num==num)
                        return true;
                else
                        return false;
        }
}
```

HashSet<E> 클래스

```
import java.util.Iterator;
import java.util.HashSet;

class SimpleNumber{
        int num;
        public SimpleNumber(int n)  {  num=n;  }
        public String toString()  {  return String.valueOf(num);  }
        public int hashCode(){
                return num%3;
        }
        public boolean equals(Object obj){
                SimpleNumber comp=(SimpleNumber)obj;
                if(comp.num==num)
                        return true;
                else
                        return false;
        }
}
```

HashSet<E> 클래스

```
class HashSetEqualityTwo
{
        public static void main(String[] args)
        {
            HashSet<SimpleNumber> hSet =
                                        new HashSet<SimpleNumber>();

            hSet.add(new SimpleNumber(10));
            hSet.add(new SimpleNumber(20));
            hSet.add(new SimpleNumber(20));

            System.out.println("저장된 데이터 수: "+hSet.size());

            Iterator<SimpleNumber> itr=hSet.iterator();

            while(itr.hasNext())   System.out.println(itr.next());
        }
}
```

TreeSet<E> 클래스의 이해와 활용

- **TreeSet<E> 클래스는 트리라는 자료구조를 기반으로 데이터를 저장한다.**
  오름차순 / 내림차순으로 가능
- **데이터를 정렬된 순서로 저장하며, HashSet<E>와 마찬가지로 데이터의 중복저장 않는다.**
- **정렬의 기준은 프로그래머가 직접 정의한다.**
  어떤기준으로 정렬할건지는 프로그래머가 정의
  ex) 나이순, 이름순, 번호순 등등

TreeSet<E> 클래스의 이해와 활용

```
public static void main(String[] args)
{
        TreeSet<Integer> sTree=new TreeSet<Integer>();
        sTree.add(1);
        sTree.add(2);
        sTree.add(4);
        sTree.add(3);
        sTree.add(2);

        System.out.println("저장된 데이터 수: "+sTree.size());

        Iterator<Integer> itr=sTree.iterator();
        while(itr.hasNext())   System.out.println(itr.next());
}
```

데이터는 정렬되어 저장이 되며, 때문에 iterator 메소드의 호출로 생성된 반복자는 오름차순의 데이터 참조를 진행한다.

출력순서가 정렬되어 있음에 주목해야 한다!
이것이 TreeSet<E>의 특징이다.

TreeSet<E> 클래스의 이해와 활용

```
import java.util.Iterator;
import java.util.TreeSet;

class SortTreeSet{
	public static void main(String[] args){
		TreeSet<Integer> sTree=new TreeSet<Integer>();
		sTree.add(1);
		sTree.add(2);
		sTree.add(4);
		sTree.add(3);
		sTree.add(2);

		System.out.println("저장된 데이터 수: "+sTree.size());

		Iterator<Integer> itr=sTree.iterator();
		while(itr.hasNext())
			System.out.println(itr.next());
	}
}
```

정렬의 기준을 정하는 Comparable<T> 인터페이스

- TreeSet<E> 인스턴스에 저장이 되려면 Comparable<T> 인터페이스를 구현해야 한다.
- Comparable<T> 인터페이스의 유일한 메소드는 int compareTo(T obj); 이다.
- compareTo 메소드는 다음의 기준으로 구현을 해야한다.

  → 인자로 전달된 obj가 작다면 양의 정수를 반환해라.

  → 인자로 전달된 obj가 크다면 음의 정수를 반환해라.

  → 인자로 전달된 obj와 같다면 0을 반환해라.

  '작다', '크다', '같다'의 기준은 프로그래머가 결정!

정렬의 기준을 정하는 Comparable<T> 인터페이스

```
class Person implements Comparable<Person>
{
    String name; int age;
    .....
    public int compareTo(Person p)
    {
        if(age>p.age)              return 1;
        else if(age<p.age)         return -1;
        else                       return 0;
    }
}
```

Person 클래스의 comareTo 메소드는 정렬의 기준을 '나이의 많고 적음'으로 구현 하였다.

정렬의 기준을 정하는 Comparable<T> 인터페이스

```
import java.util.Iterator;
import java.util.TreeSet;

class Person implements Comparable<Person>{
        String name;  int age;
        public Person(String name, int age){
                this.name=name;  this.age=age;
                }
        public void showData(){
                System.out.printf("%s %d ₩n", name, age);
        }
        public int compareTo(Person p){
                if(age>p.age)
                        return 1;
                else if(age<p.age)
                        return -1;
                else
                return 0;
        }
}
```

정렬의 기준을 정하는 Comparable<T> 인터페이스

```
class ComparablePerson
{
	public static void main(String[] args)
	{
		TreeSet<Person> sTree=new TreeSet<Person>();
		sTree.add(new Person("Lee", 24));
		sTree.add(new Person("Hong", 29));
		sTree.add(new Person("Choi", 21));

		Iterator<Person> itr=sTree.iterator();
		while(itr.hasNext())
			itr.next().showData();
	}
}
```

정렬의 기준을 정하는 Comparable<T> 인터페이스

```
import java.util.TreeSet;
import java.util.Iterator;

class MyString implements Comparable<MyString>{
        String str;
        public MyString(String str) {   this.str=str;   }

        public int getLength()   {   return str.length();        }

        public int compareTo(MyString mStr)     {
                if(getLength()>mStr.getLength())
                        return 1;
                else if(getLength()<mStr.getLength())
                        return -1;
                else
                        return 0;

                /* return getLength()-mStr.getLength(); //-1,0,1   */
        }
```

정렬의 기준을 정하는 Comparable<T> 인터페이스

```
	public String toString()
	{
		return str;
	}
}

class ComparableMyString{

	public static void main(String[] args){
		TreeSet<MyString> tSet=new TreeSet<MyString>();
		tSet.add(new MyString("Orange"));
		tSet.add(new MyString("Apple"));
		tSet.add(new MyString("Dog"));
		tSet.add(new MyString("Individual"));

		Iterator<MyString> itr=tSet.iterator();
		while(itr.hasNext())
			System.out.println(itr.next());
	}
}
```

식별 - > 중복x

Key value

Collection<E>를
상속하지 않는다.

Map(K, V) 인터페이스를 구현하는 컬렉션 클래스들

Map<K, V> 인터페이스와 HashMap<K, V> 클래스

- Map<K, V> 인터페이스를 구현하는 컬렉션 클래스는 key-value 방식의 데이터 저장을 한다.
- Value는 저장할 데이터를 의미하고, key는 value를 찾는 열쇠를 의미한다.
- Map<K, V>를 구현하는 대표적인 클래스로는 HashMap<K, V>와 TreeMap<K, V>가 있다.
- TreeMap<K, V>는 정렬된 형태로 데이터가 저장된다.

  key값으로 정렬.

Map<K, V> 인터페이스와 HashMap<K, V> 클래스

```
public static void main(String[] args)
{
        HashMap<Integer, String> hMap =
                                        new HashMap<Integer, String>();
        hMap.put(new Integer(3), "나삼번");
        hMap.put(5, "윤오번");
        hMap.put(8, "박팔번");
        System.out.println("6학년 3반 8번 학생: " +
                                          hMap.get(new Integer(8)));
        System.out.println("6학년 3반 5번 학생: "+hMap.get(5));
        System.out.println("6학년 3반 3번 학생: "+hMap.get(3));
        hMap.remove(5);                  /* 5번 학생 전학 감 */
        System.out.println("6학년 3반 5번 학생: "+hMap.get(5));
}
```

오토박싱

Map<K, V> 인터페이스와 HashMap<K, V> 클래스

```
import java.util.HashMap;

class IntroHashMap{
        public static void main(String[] args){
            HashMap<Integer, String> hMap =
                                        new HashMap<Integer, String>();
            hMap.put(new Integer(3), "나삼번");
            hMap.put(5, "윤오번");
            hMap.put(8, "박팔번");

            System.out.println("6학년 3반 8번 학생: "+hMap.get(new
Integer(8)));
            System.out.println("6학년 3반 5번 학생: "+hMap.get(5));
            System.out.println("6학년 3반 3번 학생: "+hMap.get(3));
            hMap.remove(5);                 /* 5번 학생 전학 감 */
            System.out.println("6학년 3반 5번 학생: "+hMap.get(5));


        }               // vlaue에 상관없이 중복된 key의 저장은 불가능,
}                   // vlaue는 같더라도 key가 다르면 줄 이상의 데이터도 저장이 가능
```

Map<K, V> 인터페이스와 HashMap<K, V> 클래스

```
public static void main(String[] args)
{
        TreeMap<Integer, String> tMap=new TreeMap<Integer, String>();
        tMap.put(1, "data1");
        tMap.put(3, "data3");
        tMap.put(5, "data5");
        tMap.put(2, "data2");
        tMap.put(4, "data4");
        NavigableSet<Integer> navi=tMap.navigableKeySet();
        System.out.println("오름차순 출력...");
        Iterator<Integer> itr=navi.iterator();
        while(itr.hasNext())            System.out.println(tMap.get(itr.next()));
        System.out.println("내림차순 출력...");
        itr=navi.descendingIterator();
        while(itr.hasNext())            System.out.println(tMap.get(itr.next()));
        }
```

예제에서 보이듯이 descendingIterator 메소드는 내림차순의 검색을 위한 반복자를 생성한다. 그리고 NavigableSet<E> 클래스도 Set<E> 클래스를 상속하는 컬렉션 클래스이다!

Map<K, V> 인터페이스와 HashMap<K, V> 클래스

```
import java.util.TreeMap;
import java.util.Iterator;
import java.util.NavigableSet;

class IntroTreeMap
{
        public static void main(String[] args)
        {
                TreeMap<Integer, String> tMap=new TreeMap<Integer,
String>();

                tMap.put(1, "data1");
                tMap.put(3, "data3");
                tMap.put(5, "data5");
                tMap.put(2, "data2");
                tMap.put(4, "data4");
```

Map<K, V> 인터페이스와 HashMap<K, V> 클래스

```
        NavigableSet<Integer> navi=tMap.navigableKeySet();
        // navigableKeySet 메소드가 호출되면,
        // 인터페이스 NavigableSet<E>를 구현하는 인스턴스가 반환
        // 이때 E는 key의 자료형인 Integer, 반환된 인스턴스에는 저장한 //
        // 데이터의 key 정보가 저장

        System.out.println("오름차순 출력...");
        Iterator<Integer> itr=navi.iterator();
        while(itr.hasNext())
                System.out.println(tMap.get(itr.next()));

        System.out.println("내림차순 출력...");
        itr=navi.descendingIterator();   //내림차순 정렬
        while(itr.hasNext())
                System.out.println(tMap.get(itr.next()));
    }
}
```

컬렉션 프레임웍(collection framework)이란?

▶ 컬렉션 프레임웍(collection framework)

- 데이터 군(群)을 저장하는 클래스들을 표준화한 설계
- 다수의 데이터를 쉽게 처리할 수 있는 방법을 제공하는 클래스들로 구성
- JDK1.2부터 제공

저장 , 참고 , 삭제 , 정렬 등

▶ 컬렉션(collection)

- 다수의 데이터, 즉, 데이터 그룹을 의미한다.

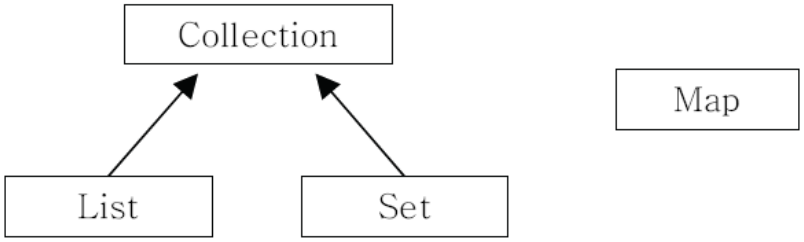
▶ 프레임웍(framework)

- 표준화, 정형화된 체계적인 프로그래밍 방식

▶ 컬렉션 클래스(collection class)

- 다수의 데이터를 저장할 수 있는 클래스(예, Vector, ArrayList, HashSet)

컬렉션 프레임웍의 핵심 인터페이스

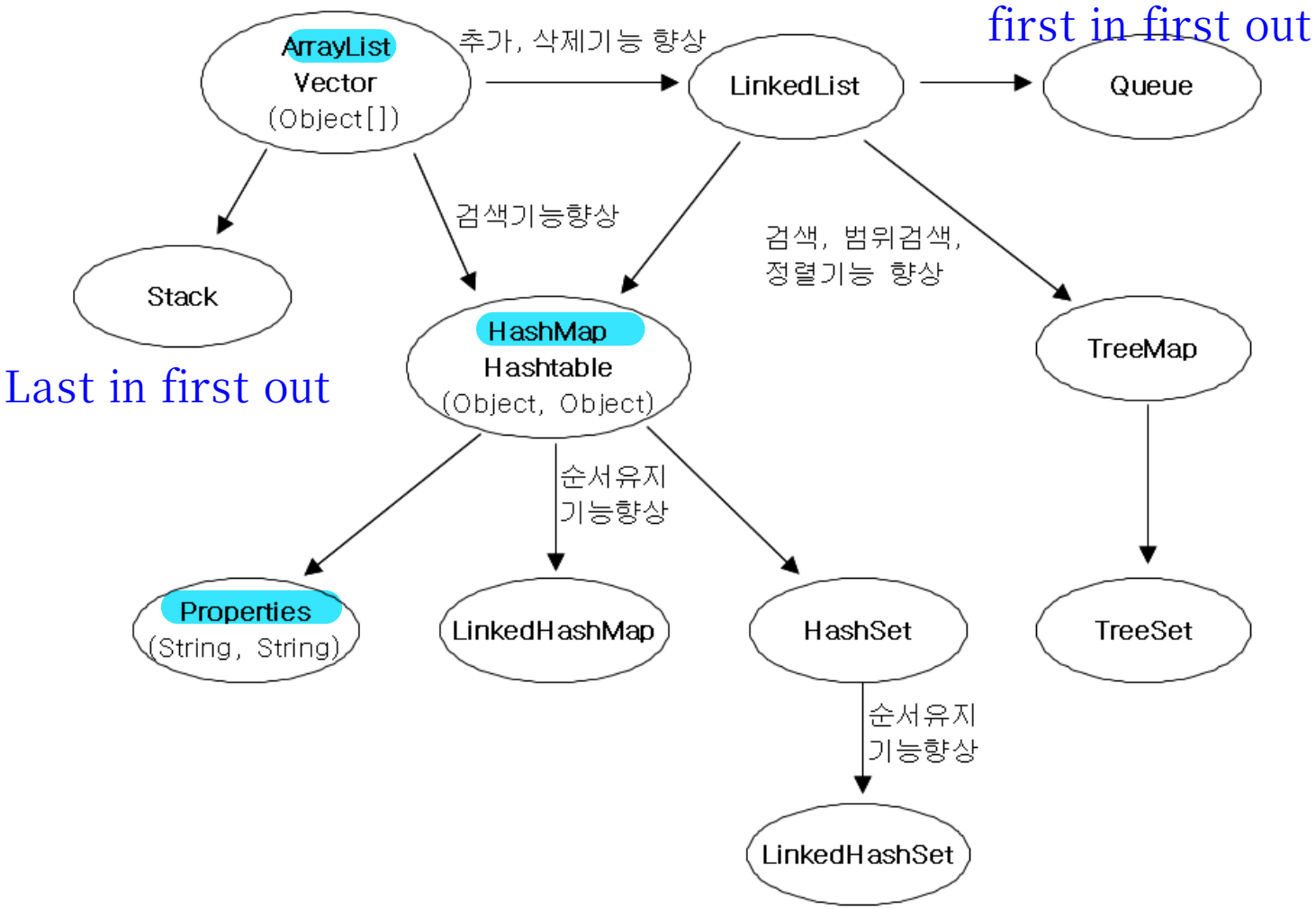


[그림11-1] 컬렉션 프레임웍의 핵심 인터페이스간의 상속계층도

인터페이스	특 징
List	순서가 있는 데이터의 집합. 데이터의 중복을 허용한다. 예) 대기자 명단
	구현클래스 : ArrayList, LinkedList, Stack, Vector 등
Set	순서를 유지하지 않는 데이터의 집합. 데이터의 중복을 허용하지 않는다. 예) 양의 정수집합, 소수의 집합
	구현클래스 : HashSet, TreeSet 등
Map	키(key)와 값(value)의 쌍(pair)으로 이루어진 데이터의 집합 순서는 유지되지 않으며, 키는 중복을 허용하지 않고, 값은 중복을 허용한다. 예) 우편번호, 지역번호(전화번호)
	구현클래스 : HashMap, TreeMap, Hashtable, Properties 등

[표11-1] 컬렉션 프레임웍의 핵심 인터페이스와 특징

컬렉션 클래스


ArrayList
Vector
(Object[])
추가, 삭제기능 향상
LinkedList
first in first out
Queue
Stack
Last in first out
검색기능향상
검색, 범위검색,
정렬기능 향상
HashMap
Hashtable
(Object, Object)
TreeMap
순서유지
기능향상
Properties
(String, String)
LinkedHashMap
HashSet
TreeSet
순서유지
기능향상
LinkedHashSet

Project

Project ver 0.60

배열을 이용해서 저장하는 방식을 ArrayList<T> 컬랙션을 이용해서 구현해 보자.

그리고 사용자 메뉴 입력시 예외처리를 해봅시다.